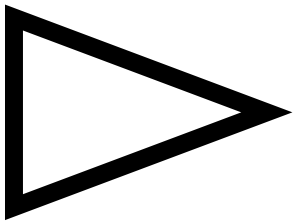

Designing better indexes and influencing DB2 on z/OS index usage

By Susan Lawson and Daniel Luksetich, YL&A

bmcsoftware
BUSINESS RUNS ON I.T.™

TABLE OF CONTENTS

INTRODUCTION

Indexes are one of the most powerful performance features of a relational database, and they are essential to support high performance applications. Proper index design and utilization is critical when designing a database with the goal of balancing costs with performance. Indexes use disk space, and they increase the cost of insert, delete, and some update operations, but they also enforce business rules, keep data organized, and speed access to the data. A proper understanding of how DB2 uses indexes is critical to the proper management of your databases and database applications. You need to know how DB2 will match predicates to index columns, and what techniques you can use to maximize index matching while minimizing the number of indexes (at least for databases that are updated). You also need to be aware of how queries are using indexes, and if you can improve the index utilization for those queries. Doing so could be the most significant performance enhancement you can provide for you DB2 applications.

For high performance DB2 applications it is important not only to understand how DB2 is optimizing its access to the data, but also how much data is being processed and retrieved at each stage of the SQL statement. Filtering as much as possible - as soon as possible - is essential to achieving optimal application performance. This paper discusses index design and query performance topics, including best use of index matching predicates, index screening, and influencing DB2 to use particular indexes.

INDEX UTILIZATION IN DB2 FOR Z/OS

A DB2 index is a list of the locations of rows sorted by the contents of one or more specified columns. We typically use indexes to improve query performance. However, indexes can also serve a logical data design purpose. For example, a unique index does not allow the entry of duplicate values in columns, thereby guaranteeing that no rows of a table are the same. You can create indexes to specify ascending, descending, or random order by the values in a column. The indexes contain a pointer, known as a record ID (RID), to the physical location of the rows in the table.

REASONS TO USE INDEXES

There are three main reasons to create indexes:

- To improve query performance
- To ensure uniqueness of values
- To ensure a physical clustering sequence of table data

You can create more than one index on a particular base table. However, the more indexes you have, the more the database system must work to keep the indexes current during update, delete, and insert operations. Creating a large number of indexes for a table that receives many changes can slow processing, and a large number of indexes can lead to longer outages during maintenance work like reorganizations or loading for data.

Indexes consume disk space. The amount of disk space varies depending on the length of the key columns, compression option, and whether the index is unique or non-unique. Index size and levels increases as you add more data into the base table. Therefore, consider the disk space required for indexes when planning the size of the database. Some of the key index design rules are:

- Primary and unique key constraints require a unique index.
- PCTFREE and FREEPAGE can greatly help insert performance.
- Create a clustering index if you need to read ranges of data to utilize sequential prefetch.
- Create indexes on foreign key constraint columns to speed certain operations

TYPES OF INDEXES

Let's look at some types of indexes in DB2 and discuss some best practices. This paper does not cover all index types, such as XML and LOB indexes, but it discusses index techniques that can enhance performance for your SQL statements.

CLUSTERING

In general, it's not that important to control the physical sequence of the data in a table. But to utilize sequential reads for range processing through an index, use the CLUSTER option on one, and only one, index on a table to specify the physical sequence. Without this option, the first index defined on the table in a non-partitioned tablespace is used for the clustering sequence. Be careful when you have multiple indexes on a table, and don't rely on that 'first' index for clustering, because if you need to drop and recreate this first index, another index will become the 'first' index and the clustering in the table will be changed during the reorg, load or recover of the tablespace.

The best clustering index is one that supports the majority of the sequential access to the data in the table. For example, if a large batch process reads data based on an input key in a specific order, it may be best to make the column corresponding to that input sequence the clustering key. Tables that are often accessed together in the same process can benefit from being clustered on a common value, which can promote the use of sequential detection and index lookaside, as well as minimize random I/O operations.

PARTITIONED

To create an index that is partitioned according to the partitioning scheme of the underlying data, specify the PARTIONED keyword on the CREATE INDEX statement. Two types of partitioned indexes are available: partitioning and secondary.

For an index to be considered a partitioning index , the specified index key columns must match or comprise a superset of the columns specified in the partitioning (limit) key, must be in the same order, and must have the same ascending or descending attributes. If an index on a partitioned tablespace does not have the attributes of a primary partitioning index, it is considered a secondary index.

When table-controlled partitioning is in place, the index definition simply needs to include the PARTITIONED keyword to indicate that the index is a partitioned index. Just like the partitioned tablespace, the partitioned index consists of several data sets. Each partition can have different attributes (some may have more free space than others).

NON-PARTITIONING SECONDARY INDEX (NPSI)

NPSIs are indexes that are used on partitioned tables. They are not the same as the partitioning key, which is used to order and partition the data; rather, they improve access to the data. While partitioning and partitioned indexes have one index partition per tablespace partition, one NPSI will reference the entire tablespace, spanning all partitions. NPSIs can be unique or non-unique.

You can break NPSIs into multiple pieces (data sets) by using the PIECESIZE clause on the CREATE INDEX statement. Pieces can be 254 K to 64 GB; the best size will depend on how much data you have and how many pieces you want to manage. If you have several pieces, you can achieve more parallelism on processes such as heavy INSERT batch jobs by alleviating the bottlenecks caused by contention on a single data set. The following example shows how to create an NPSI with 1M pieces.

```
CREATE UNIQUE INDEX DSN8910.XEMP9
ON DSN8910.EMP
(LASTNAME ASC)
PADDED
USING STOGROUP DSN8G910
PRIQTY 512
SECQTY 64
ERASE NO
BUFFERPOOL BP1
CLOSE YES
PIECESIZE 1M;
```

DATA PARTITIONED SECONDARY INDEXES (DPSI)

The data partitioned secondary index (DPSI) provides many advantages for secondary indexes on a partitioned tablespace over the traditional NPSIs in terms of availability.

The partitioning scheme of the DPSI will be the same as the tablespace partitions, and the index keys in index partition 'x' will match those in tablespace partition 'x'..

DPSIs provide the following benefits:

- Optional clustering by a secondary index
- Ability to easily rotate partitions
- Efficient utility processing on secondary indexes
- Potential reduced overhead in data sharing (affinity routing)

Although DPSIs further improve partition independence and reduce contention for concurrent utilities, some queries may not perform as well. Queries with predicates that reference columns in a single partition and are therefore restricted to a single partition of the DPSI will benefit from this new organization. To achieve this benefit, design the queries to allow for partition pruning through the predicates: in the query, supply at least the leading column of the partitioning key so that DB2 can eliminate partitions from the query access path. If a predicate references only columns in the DPSI, the SQL statement may not perform very well because it may need to probe several partitions of the index.

Other limitations to using DPSIs include the fact that they cannot be unique (in DB2 9, they can be unique within the partition) and they may not be the best candidates to support ORDER BYs.

UNIQUE INDEX AND NON-UNIQUE INDEX

A unique index guarantees the uniqueness of the data values in a table's columns. The uniqueness is enforced at the end of the SQL statement that updates rows or inserts new rows. The uniqueness is also checked during execution of the CREATE INDEX statement. If the table already contains rows with duplicate key values, the index is not created. A primary partitioned index can be unique; a secondary partitioned index can only be unique within the partition (as of DB2 9).

INDEX ON EXPRESSION

An index on expression allows you to use scalar expressions in place of columns in the definition of a unique or non-unique index. Expressions are results derived from calculations and/or functions. You can specify a key-expression in place of a column list in a CREATE INDEX statement. DB2 can then use the index on the expression to match predicates in SQL statements that include the expression. The key-expression must contain at least one column from the referenced table that is not a LOB, XML, or DECFLOAT column. In addition, the key-expression must not include:

- A subquery
- An aggregate function
- A user-defined function
- A sequence object reference
- A special register
- A CASE expression

If the index was defined as unique, the uniqueness will be enforced on the result of the expression.

Below is an example of using an index on expression with a SOUNDEX function:

```
CREATE TYPE 2 INDEX DSN8910.XEMP3
       ON DSN8910.EMP

       (cast(soundex(lastname) as char(4) ccsid ebcdic)  ASC  )
       USING STOGROUP DSN8G910
             PRIQTY 12
             ERASE NO
       FREEPAGE 0
       PCTFREE 10
       BUFFERPOOL BP0
       CLOSE NO
       PIECESIZE 2097152  K;
```

DB2 considers the index on an expression during access path selection. It's possible to collect statistics on the expression; system catalog tables SYSKEYTARGETS, SYSKEYTARGETSTATS, SYSKEYTGTDIST, and SYSKEYTGTDISTSTATS support these statistics.

HOW DB2 MATCHES PREDICATES

The most expensive thing we can do when processing DB2 is call DB2 from the application. Once we have made this initial call, it is important to process as efficiently as possible. This means filtering the data as close to the indexes and data as possible. When coding or tuning SQL statements, remember this - filter as much as possible as early as possible. This means exploiting those indexes we created.

When DB2 processes a predicate (WHERE clause) in the SQL statement, it does it in various stages of the DB2 engine. Predicates in SQL statements fall under different classifications that dictate how DB2 processes the predicates and how much data is filtered during the process. The classifications are:

- Stage 1(indexable)
- Stage 1 (non-indexable)
- Stage 2

The DB2 Stage 1 engine understands your indexes and tables, and it can use an index for efficient access to your data. Only a Stage 1 predicate can limit the range of data accessed on a disk. The Stage 2 engine processes functions and expressions, but it is not able to directly access data in indexes and tables. Data from Stage 1 is passed to Stage 2 for further processing. Therefore, Stage 1 predicates are generally more efficient than Stage 2 predicates. Stage 2 predicates cannot use an index, and thus cannot limit the range of data retrieved from disk. Finally, some predicates are processed outside of DB2 in the application layer (the application discards or ignores data after it is retrieved from DB2). In this case, the filtering is performed after the data is retrieved from DB2 and processed in the application. These application predicates are the least efficient.

STAGE 1 INDEXABLE

Stage 1 indexable predicates provide for the best filtering and performance. However, many additional factors, such as syntax and length/type of constants used, determine actual index usage and stage of processing. All indexable predicates are Stage 1, but not all Stage 1 predicates are indexable.

Stage 1 indexable predicates are predicates that can be used to match on the columns of an index. The most simple example of a Stage 1 predicate would be of the form <col op value>, where col is a column of a table, op is an operator (=, >, <, >=, <=), and value represents a non-column expression (an expression that does not contain a column from the table). Predicates containing BETWEEN, IN (for a list of values), and LIKE (without a leading search character) can also be Stage 1 indexable. When an index is used, the Stage 1 indexable predicate provides the best filtering because it can actually limit the range of data accessed from the index. Use Stage 1 matching predicates whenever possible. Please see the *IBM DB2 Performance Monitoring and Tuning Guide* for a complete list of Stage 1 indexable predicates.

Below is an example of a Stage 1 indexable (assuming we have an index on DEPTNO) predicate:

```
SELECT DEPTNO
 FROM DEPT
 WHERE DEPTNO  = 'A01'
```

STAGE 1 NON-INDEXABLE

Not all Stage 1 predicates are indexable. Some Stage 1 predicates will not be usable for index access. These predicates are generally of the form <col NOT op value>, where col is a column of a table, op is an operator, and value represents a non-column expression, host variable, or value. Predicates containing NOT BETWEEN, NOT IN (for a list of values), NOT LIKE (without a leading search character), or LIKE (with a leading search character) can also be Stage 1 indexable. Although non-indexable Stage 1 predicates cannot limit the range of data read from an index, they are available as index screening predicates. Please see the IBM DB2 Performance Monitoring and Tuning Guide for a complete list of Stage 1 non-indexable predicates.

Below is an example of a Stage 1 non-indexable (assuming we have an index on DEPTNO) predicate:

```
SELECT DEPTNO
 FROM DEPT
 WHERE DEPTNO <> 'A01'
```

Remember that a Stage 1 indexable predicate is the best we can get for most filtering, so think positively when coding your predicates. Negative logic (NOT) is usually not indexable.

STAGE 2

Stage 2 predicate evaluation happens after the data accesses and performs activities such as sorting, function evaluation, and expression evaluation. Data access cannot be limited by Stage 2 predicates, therefore these predicates are generally more expensive than Stage 1 predicates because they are evaluated later in the processing and require more data movement in the DB2 engine.

Stage 2 predicates are generally the more complex predicates. This includes predicates that contain items such as column expressions, correlated subqueries, and CASE expressions. A predicate can also appear to be Stage 1, but processed as Stage 2, for instance, any predicate processed after a join operation is Stage 2.

DB2 does a pretty good job of promoting mismatched data types to Stage 1 via casting, however some predicates with mismatched data types will remain Stage 2.

An example of a Stage 2 predicate is shown below. Note that what makes this a Stage 2 predicate is the fact that we are comparing an expression to a literal. This is a common predicate type, but it is not the most efficient, hence the reason we have index on expression in DB2 9.

```
WHERE SUBSTR(LASTNAME,1,1) = 'L'
```

APPLICATION LAYER PREDICATES

An application predicate is code that filters data after the data has been retrieved from DB2. Basically, it is filtering done in the application program. This is not a desirable method for coding against DB2. Performance is best served when all filtering is done in DB2, and not in the application code. Try to keep any filtering logic in the SQL statement.

Often many application layer predicates are present, but they are not seen by those who are tuning SQL. Why? Because you are looking at SQL! When tuning any application, we must look at all the code to be sure we are using predicates properly and not doing the majority of filtering (and joining!) in the application code. Generic I/O module applications are probably the worst culprits.

BOOLEAN TERM PREDICATES

A Boolean term predicate is a simple or compound predicate that, when evaluated false for the row, makes the entire WHERE clause evaluate to false. Boolean term predicates can only utilize single index access. This is an important note for SQL performance. Non-Boolean term predicates may be considered for multi-index access. The following compound predicate is Boolean term because either individual predicate makes the entire WHERE clause false if they are false.

```
WHERE LASTNAME = 'JONES' AND FIRST_INIT = 'T'
```

This predicate can use an index on LASTNAME or and index on FIRST_INIT if either of those indexes exist. The following compound predicate is non-Boolean term and can only use muti-index access for each individual predicate if both of those indexes exist.

```
WHERE LASTNAME = 'JONES' OR FIRST_INIT = 'T'
```

INFLUENCING INDEX SELECTION

Having accurate catalog statistics and coding proper predicates gives DB2 an excellent opportunity to properly choose the right index for each query. In some situations, however, we may know more than DB2. For example, we know common input values, the organization on input files to processes, and the sequence of statements within a process. Knowing these things, we may want to influence DB2 to use a particular index.

CATALOG STATISTICS

We cannot overstate the importance of proper catalog statistics on the ability of the DB2 Optimizer to choose the correct access path and best index in support of the access path. However, what exactly is the definition of proper catalog statistics? This depends upon several factors, including how well you know your data, how well you know the processes accessing the data, what type of SQL is executing, how often that SQL executes, the organization of the data, and REORG frequency. That is certainly a significant amount of information to discern, but we can break it down to some basics.

The DB2 system catalog can store cardinality statistics, frequency distribution statistics, and histogram statistics. If your applications contain primarily SQL that uses host variables and parameter markers in predicates, then for the most part cardinality statistics are good enough. DB2 looks at the cardinality (number of distinct values) of columns in the tables referenced in your SQL statements to determine the filter factor (ability to restrict rows returned) for various predicates. This influences the Optimizer on access path selection and index selection. If your SQL statements include embedded literal values, then perhaps distribution and/or histogram statistics are

more appropriate. With distribution statistics, DB2 stores least or most frequent column values and the percentage that these values occur. This allows DB2 to analyze values embedded in SQL statements to more accurately determine the filter factor based upon those values compared to the frequent values. Histogram statistics contain groups of ranges of values, called quantiles, spanning the entire domain of values for a column and the percentage of values within this range. These types of statistics are best in support of range predicates with embedded literal values, but are also good in some cases for non-range predicates with embedded literals for diverse ranges of inputs.

Column correlation statistics can also be gathered when two or more columns might be correlated (common sets of values occur together). DB2 can store cardinality information for combinations of columns in the system catalog. This is especially useful if you have sets of predicates that are common in your queries.

So, to get the best index utilization and access paths should you collect the most statistics possible? Maybe. Collecting these statistics can be expensive, so you need to balance the amount you collect with their usefulness in your environment. The quantity of available statistics can influence SQL statement compile time. This means that bind or prepare costs can increase, depending on the amount of information the Optimizer must process. You should definitely collect and maintain catalog statistics, depending upon what type of statements exist and how often they execute.

PROPER PREDICATES AND PROMOTION

There are limits to the ability of the Optimizer to match predicates to index columns. For example, expressions typically are non-indexable Stage 2 predicates. If you code the following expression:

```
WHERE SUBSTR(LASTNAME,1,1) = 'L'
```

then DB2 cannot use an index on LASTNAME to match the predicate unless you've built an index on that exact expression. You can improve index matching for your predicates. For example, you can promote predicates if you employ some simple rules when coding. One such rule would be to avoid expressions against indexable columns in predicates. Where the previous predicate is looking for last names beginning with the letter "L" is not indexable, the following Stage 1 predicate is:

```
WHERE LASTNAME LIKE 'L%'
```

The IBM DB2 Performance Monitoring and Tuning Guide contains a guide to indexable versus non-indexable predicates. Range predicates can be indexable, but predicates following range predicates cannot. For example, the following predicate contains an indexable range predicate on department, but the predicate following it is not indexable:

```
WHERE WORKDEPT BETWEEN 'A20' AND 'A22'
AND LASTNAME LIKE 'L%'
```

If you know something about your data, such as department numbers are actually one alphabetic followed by two numerics, then you can use that information in combination with the information about indexability from the performance guide, and code a compound predicate that has a better chance of matching on multiple columns (if a supporting index exists, of course).

```
WHERE WORKDEPT IN ('A20', 'A21', 'A22')
AND LASTNAME LIKE 'L%
```

If ranges are applied in predicates to columns with limited domains, you could use a technique that we like to call "building an index on the fly." For example, imagine that a DB2 sample database 100,000 employees and 100 departments. If we needed to search for employee names beginning with "L" within 20 departments, we may not get an index match on the LASTNAME predicate (with a supporting index on WORKDEPT, LASTNAME). The IN predicate on 20 values also may not get us matching index access on both columns. However, with an index on the DEPTNO in the DEPT table, we can apply the range there and then join to the EMP table by IN subquery or JOIN syntax. The idea is to apply the range to a small table

(100 departments) and an equals predicate to the large table (100,000 employees). This query does that:

```
SELECT E.*
FROM DEPT D
INNER JOIN
EMP E
ON D.DEPTNO = E.WORKDEPT
WHERE D.DEPTNO BETWEEN 'A01' AND 'A20'
AND E.LASTNAME LIKE 'L%'
```

Here is another example: :

```
SELECT E.*
FROM EMP E
WHERE E.WORKDEPT IN
(SELECT D.DEPTNO
 FROM DEPT D
 WHERE D.DEPTNO BETWEEN 'A01' AND 'A20')
AND E.LASTNAME LIKE 'L%'
```

This index building technique can be extremely useful in improving index utilization without increasing the number of indexes supported.

INFLUENCING THE OPTIMIZER TO CHOOSE AN INDEX

Besides the techniques mentioned above, we have several tricks you can use to influence DB2 to choose one index over another. Some work quite well, and others work only sometimes so be sure to test using EXPLAIN and benchmark executions.

» Code an ORDER BY on index columns in index column sequence

DB2 will use an index to avoid a sort in support of an ORDER BY clause if it can. If the Optimizer has to choose between more than one potential index and you want a certain index chosen, use this technique to pick the index you want.

» Add index columns to make the query index only

If the Optimizer is not choosing the index you want it to, you could, if it's feasible, move all the columns from the table into the desired index. This could be the turning point as DB2 may choose the index over others in order to avoid the table access.

» Increase index match columns

You can influence DB2 to choose a particular index if you promote predicates, change the order of columns in the index, add/or remove columns from the index, or add predicate enablers. A predicate enabler is a predicate you add to the query that does not provide additional filtering, but could encourage matching. For example AND SEX IN ('M','F') could be a predicate enabler as well as AND BIRTH_DTE >= '0001-01-01'.

» Disable predicates on other indexable columns

Certain non-indexable predicates, when connected to an indexable predicate by an OR, can render that predicate non-indexable. For example, if you have AND WORKDEPT = 'D01' and you want to disable indexability and encourage the use of a different index, then you could code AND (WORKDEPT = 'D01' OR 0=1). Use predicate disablers cautiously and only as a last resort.

» Code an OPTIMIZE FOR clause

The OPTIMIZE FOR clause tells DB2 how many rows you will use from a query. Coding a specific value could cause the Optimizer to choose another index in support of a different access path. OPTIMIZE FOR 1 ROW is the strongest encouragement.

» Make the table VOLATILE

VOLATILE directs DB2 to use index access to the table whenever possible. However, list prefetch and certain other optimization techniques are disabled when VOLATILE is used. If you add the VOLATILE option to a table, DB2 could be encouraged to choose an index in spite of what catalog statistics may indicate. Use with caution.

» Code an equality in a subquery to discourage index access by transitive closure

During query transformation, DB2 can write additional predicates based on transitive closure. For example, if you join the DEPT table to the EMP table using the predicate "ON D.DEPTNO = E.WORKDEPT" and a local predicate "WHERE D.DEPTNO = 'D01'", then DB2 will add "AND E.WORKDEPT = 'D01'" and this may influence index access. To limit this possibility, code the equals on a subquery instead of a literal value, such as "WHERE E.DEPTNO = (SELECT 'D01' FROM SYSIBM.SYSDUMMY1)". Subqueries are not available for transitive closure.

USING CORRELATION OR NON-CORRELATION TO MAXIMIZE INDEX UTILIZATION

We typically use correlation to encourage or discourage index access for subqueries and joins. Correlated references encourage DB2 to use index access for the correlated references when a supporting index is available. This is a specialized technique, and you need to have sufficient knowledge about your data to use it effectively.

You can code any subquery as a correlated subquery, non-correlated subquery, or a join. DB2 can convert subqueries into any of these constructs during query transformation. So, no matter how you code a subquery, DB2 may recode it for you anyway. Therefore, it is critical to check the output from an EXPLAIN to see what DB2 has done with your query.

The following query might be the better choice if you have an index on the WORKDEPT column of the EMP table and few qualifying rows for the DEPT table.

```
SELECT *
FROM DEPT D
WHERE MGRNO = ?
AND EXISTS
(SELECT 1
FROM EMP E
WHERE E.WORKDEPT = D.DEPTNO
AND E.SALARY > 100000)
```

In contrast, the following logically equivalent statement may be better if you have no index on the WORKDEPT column, many qualifying rows in the DEPT table, and an index on the DEPTNO column of the DEPT table:

```
SELECT *
FROM DEPT D
WHERE MGRNO = ?
AND D.DEPTNO IN
(SELECT E.WORKDEPT
FROM EMP E
WHERE E.SALARY > 100000)
```

With some joins, especially into views and table expressions, indexability in support of the join may be limited. DB2 may materialize the view or table expression first, and then join. You could use a correlated table expression to force index access in these cases. However, be extremely careful with this technique as EXPLAIN output may show a better access path, but the query may use more CPU. For this reason, you should benchmark test every query change.

The following query will not use an index on the WORKDEPT column due to materialization of the table expression:

```
SELECT *
FROM DEPT D
LEFT OUTER JOIN
(SELECT WORKDEPT, AVG(SALARY)
FROM EMP E
GROUP BY WORKDEPT) AS X
ON D.DEPTNO = X.WORKDEPT
WHERE D.DEPTNO = 'D01'
```

The following correlated table expression can use an index on the WORKDEPT columns, but the table expression is executed once for every qualifying DEPT row:

```
SELECT *
FROM DEPT D
LEFT OUTER JOIN
TABLE (SELECT WORKDEPT, AVG(SALARY)
FROM EMP E
WHERE E.WORKDEPT = D.DEPTNO
GROUP BY WORKDEPT) AS X
ON D.DEPTNO = X.WORKDEPT
WHERE D.DEPTNO = 'D01'
```

TUNING INDEXES WITH BMC SOLUTIONS

BMC SQL Performance for DB2 helps you diagnose DB2 SQL performance problems and write applications that perform much more efficiently, thereby reducing the overall cost for running the application. BMC SQL Performance for DB2 integrates the functionality of BMC SQL Explorer for DB2 (plan analyzer capability and access path compare) and BMC APPTUNE for DB2 (SQL monitor capability including Explain analysis) into a single offering that helps you optimize performance and availability by tuning the application SQL or physical database design.

BMC SQL Performance for DB2 includes the Index advisor to ensure that existing indexes are optimally structured to support the production applications. The Index advisor provides on-demand, dynamic data collection of index dependencies and catalog statistics.

The Index Advisor extends the capability of BMC APPTUNE object analysis by collecting and reporting on column usage data for SQL statements. It automatically collects and displays actual access counts for each unique SQL statement (table and index, and predicate usage frequencies). For example, you can generate a report, which shows how many distinct SQL statements have used a particular column in any kind of predicate, or ORDER BY clause. This type of information tells you if statements access non-indexed columns or how changes to existing indexes would affect other statements in your workload. Other table and index reports provide quick access to listings of the most-used objects based on get page volume or index access ratio.

The Index Advisor also extends the capability of the Explain function by comparing access paths after making changes to virtual indexes in a cloned database. This "what-if?" index analysis lets you model changes to indexes without impact of the production environment.

BMC SQL Performance for DB2 enables you to obtain accurate, real-time performance information about DB2 indexes. Because the Index advisor presents data at the object level, you can review the index access data to evaluate the performance of your SQL and identify candidates for index improvements.

Another important factor in index tuning is to understand if indexes are used at all, and how expensive particular indexes are to your business. Performance Advisor technology in BMC SQL Performance for DB2 provides a performance management database that serves as a warehouse for performance trending and analysis of how performance changes over time. Packaged reports tell you which indexes are not used or just rarely used.

In conclusion, the Index Advisor:

» Ensures that new or changed indexes will work in production. No more firefighting of slow performing SQL after a production rollout
» Increases your productivity by simulating index changes before implementing them in production.
» Shows how your SQL workload accesses index columns of particular tables so that you can design better indexes that will reduce the cost of DB2 applications. Better DB2 performance results in better service to the business and end users.

ABOUT THE AUTHORS
Susan Lawson and Dan Luksetich are internationally recognized DB2 consultants. They specialize in DB2 performance and availability. They have both authored several articles and books on DB2. For more information or to contact them, visit www.db2expert.com.

141827